OCR Computer Science A Level

2.2.1 Programming Techniques

Advanced Notes

Specification:

2.2.1 a)

- **Programming constructs**
  - Sequence
  - Iteration
  - Branching

2.2.1 b)

- **Recursion**
  - How it can be used
  - How it compares to an iterative approach

2.2.1 c)

- **Global and local variables**

2.2.1 d)

- **Modularity, functions and procedures**
  - Parameter passing by value
  - Parameter passing by reference

2.2.1 e)

- **Use of an IDE to develop / debug a program**

2.2.1 f)

- **Use of object-oriented techniques**

Programming Constructs

A crucial part of solving a problem is simplifying it to represent it in a way that makes it easier to understand and thus program. The following constructs are used to represent a program's control flow in a popular subsection of procedural programming called structured programming:

- Sequence

  Code is executed line-by-line, from top to bottom.

- Branching

  A certain block of code is run if a specific condition is met, using IF statements. This is also known as 'selection'.

- Iteration

  A block of code is executed a certain number of times or while a condition is met. Iteration uses FOR, WHILE or REPEAT UNTIL loops.
  Iteration can be either:

  - Count-controlled

    Iteration is repeated a given number of times.

```
for i in range (0,10):
     print i
next i
```

  - Condition-controlled

    Iteration continues until a given condition is met.

```
while i <= 20:
     print "Not true";
     i=i+1
endwhile
```

Recursion

Recursion is a programming construct in which a subroutine calls itself during its execution. This continues until a certain condition - called the stopping condition - is met, at which point the recursion stops. Recursion produces the same result as iteration, but is more suited to certain problems which are more easily expressed using recursion.

Synoptic Link

Recursion is an example of the **divide and conquer** technique which is discussed further in **2.2.2.**

The advantage of using recursion for certain problems is that they can be represented in fewer lines of code, which makes them less prone to errors. However it is essential that recursive subroutines are clearly defined so as to reach a stopping condition after a finite number of function calls.

A common example of a naturally recursive function is factorial, shown below:

```
function factorial(number)
     if number == 0 or 1:
          return 1
     else:
          return number * factorial(number - 1);
     endif
end function
```

Each time the function calls itself, a new stack frame is created within the call stack, where parameters, local variables and return addresses are stored. This information allows the subroutine to return to that particular point during its execution. A finite number of stack frames are created until the stopping condition, or base case, is reached at which point the subroutine unwinds. This refers to the process of information from the call stack being popped off the stack.

Call Stack

A 'Last In First Out' data structure which stores details about the order in which subroutines have been called

There are some disadvantages to recursion, the biggest being its inefficient use of memory. If the subroutine calls itself too many times, there is a danger of a stack overflow, which is when the call stack runs out of memory. This would cause the program to crash. Tail recursion is a form of recursion which is implemented in a more efficient way in which less stack space is required. Another problem with recursion is that it is difficult to trace, especially with more and more function calls.

Global and Local Variables

Variables can be defined with either global or local scope. Scope refers to the section of code in which the variable is available.

Local variables have limited scope which means that they can only be accessed within the block of code in which they were defined. If a local variable is defined within a subroutine, it can only be accessed within that subroutine. Therefore, multiple local variables with the same name can exist in different subroutines and will remain unaffected by each other. Using local variables is considered to be good programming practice because it ensures subroutines are self-contained, with no danger of variables being affected by code outside of the subroutine.

Synoptic Link

The idea of **self-contained subroutines** follows the principle of **encapsulation** in **object-oriented programming**, encountered in **1.2.4**.

Global variables, on the other hand, can be accessed across the whole program. All variables used in the main body of a program are automatically declared to be global. These are useful for values that need to be used by multiple parts of the program. On the whole, however, using global variables is not recommended because they can be unintentionally overwritten and edited. As global variables are not deleted until the program terminates, they require more memory than local variables which are deleted once the subroutine has been completed.

In the event that a local variable exists within a subroutine with the same name as a global variable, the local variable will take precedence.

Modularity, Functions and Procedures

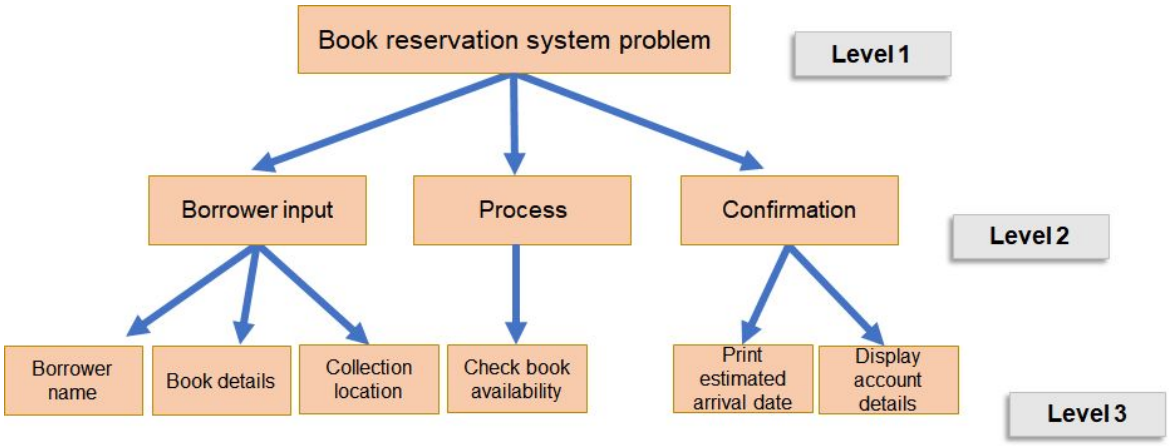
Modular programming is a programming technique used to split large, complex programs into smaller, self-contained modules. Modularity is essential to making a problem easier to understand and approach. A modular design also makes it easier to divide tasks between a team and manage, whilst simplifying the process of testing and maintenance, as each component can be dealt with individually. This improves the reusability of components, as once a module has been tested, it can be reused with confidence.

A popular technique used to modularise programs is called the top-down approach, in which the problem is continually broken down into sub-problems, until each can be represented as an individual, self-contained blackbox which performs a certain task.

Synoptic Link

Top-down design is an example of **problem decomposition** discussed in **2.2.2**.

This process is also known as stepwise refinement. These modules form blocks of code called subroutines, which can be categorised as either functions or procedures. Below is an example of a top-down design for a problem involving a book reservation system:



Procedures and functions are both named blocks of code that perform a specific task. While procedures do not have to return a value, functions must always return a value. Procedures can return multiple values whereas a function must return one, single value. Procedures are typically given data as parameters for manipulation while functions commonly make use of local variables.

Parameters

Values passed into a function.

The subroutine below is an example of a function as it always returns a value of either True or False regardless of the input.

```
function isEven(number):
     if number MOD 2 = 0:
          return True
     else:
          return False
end function
```

When parameters are passed into a subroutine, they can be passed either by value or by reference. When a parameter is passed by value, it is effectively treated as a local variable; a copy of the value is passed to the subroutine and discarded at the end therefore its value outside of the subroutine will not be affected. Passing by reference means that the address of the parameter is given to the subroutine, so the value of the parameter will be updated at the given address.

In exam questions, you should assume parameters are passed by value unless you are told otherwise. The following format will be used:

```
function multiply(x:byVal, y:byRef)
```

Use of an IDE

An Integrated Development Environment, or IDE, is a program which provides a set of tools to make it easier for programmers to write, develop and debug code. Examples of IDEs include PyCharm, Eclipse, IDLE and Microsoft Visual Studio. Common features of IDEs include:

- Stepping
  This allows you to monitor the effect of each individual line of code by executing a single line at a time.

- Variable watch
  Sometimes used to pinpoint errors, this is a useful feature to observe how the contents of a variable change in real-time through the execution of a program.

- Breakpoint
  IDEs allow users to set a point in the program at which the program will stop. This can either be based on a condition or set to occur at a specific line. This can help to pinpoint where an error is occurring.

- Source code editor
  The editor aims to make the coding process easier by providing features such as autocompletion of words, indentation, syntax highlighting and automatic bracket completion.

- Debugging tools
  Some IDEs also provide run-time detection of errors with a guide as to where in the code they are likely to have occurred through line numbers and highlighting.

Use of object-oriented techniques

The techniques used in object-oriented programming are fundamental throughout all of computer science.

Object-oriented languages are built around the idea of classes. A class is a template for an object and defines the state and behaviour of an object. State is given by attributes which give an object's properties. Behaviour is defined by the methods associated with a class, which describe the actions it can perform.

Classes can be used to create objects by a process called instantiation. An object is a particular instance of a class, and a class can be used to create multiple objects with the same set of attributes and methods.

In object-oriented programming, attributes are declared as private so can only be altered by public methods. This is called encapsulation. Encapsulation is a technique used throughout programming to implement the principle of information hiding. This is when programs are made less complex by protecting data from being accidentally edited by other parts of the program. Top-down design implements the same principle of encapsulation, as each module is designed to be self-contained.